Application Data Prefetching on the IBM Blue Gene/Q Supercomputer

I-Hsin Chung Changhoan Kim
Hui-Fang Wen Guojing Cong
IBM Research
Yorktown Heights, NY, USA
{ihchung,kimchang,hfwen,gcong}@us.ibm.com

***Abstract*—Memory access latency is often a crucial performance limitation for high performance computing. Prefetching is one of the strategies used by system designers to bridge the processor-memory gap. This paper describes a new innovative list prefetching feature introduced in the IBM Blue Gene/Q supercomputer. The list prefetcher records the L1 cache miss addresses and prefetches them in the next iteration. The evaluation shows this list prefetching mechanism reduces data fetching time when L1 cache misses happen and improves the performance for high performance computing applications with repeating non-uniform memory access patterns. Its performance is compatible with classic stream prefetcher when properly configured.**

I. Introduction

Rapidly moving data from memory through levels of cache to processor cores is a major challenge for achieving high performance in computer system design. Typical approaches to improving performance include increasing memory bandwidth, caching and prefetching. Perhaps one of the most difficult areas in optimizing memory access is when applications need to use indirect addressing. In its simplest form, this can be characterized as having an array index itself be another array of unordered index possibilities, or symbolically $A(B(j))$, where A and B are arrays and j is an ordered index. Even though j is ordered in value, the array B values are generally not, and the result is that the accesses of array A are not sequentially ordered. Such indirect addressing structures are prevalent in a wide variety of applications, many of which are scientific and engineering applications that utilize sparse matrix solution methods.

The performance effect of indirect addressing can be quite poor, depending on just how much memory inefficiency occurs. Most modern computing systems depend on efficient memory utilization in order to overcome the disparity in processor speed and memory access capability, which can be an order of magnitude or more. Furthermore, modern computer architectures implement complex caching structures that for many applications can offset this processor-memory disparity, as long as the memory accesses are fairly regular. Unfortunately, the process of indirect addressing can result in widely varied memory accesses, thus mitigating any of these architectural advantages.

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 2012 IEEE

The IBM Blue Gene/Q (BG/Q) supercomputer is a new system that introduces new technology to improve overall resource efficiency for applications that would have limited performance on more traditional architectures. The specific hardware feature that is investigated in this paper is called L1P, which is shorthand for "Level One Cache Prefetch". The L1P predicts memory access patterns using various algorithms and prefetches the data accordingly so that there is little to no delay in feeding the processor with data from memory. The L1P consists two prefetch engines: stream prefetcher and list prefetcher. The stream prefetcher handles sequential memory access patterns and the new list prefetcher essentially deals with non-sequential memory access patterns in the applications. Such a technique not only allows for performance improvement when the data streams are linear, but it also enhances performance when the memory access patterns are not linearly sequential as in the case of indirect addressing.

The remainder of this paper is organized as follows. Section II provides a summary of related work. Section III will describes the technical details of the new L1P feature of Blue Gene/Q. Section IV evaluates the efficiency of the L1P feature for selected applications. Lastly, our discussion of these results and our conclusions are presented in sections V and VI, respectively.

II. Related Work

Data prefetching has been recognized as an effective technique to hide the long memory latency for workloads with regular (e.g., strided) memory access pattern. The approaches which have been proposed range from inserting compiler-based prefetchers to pure hardware-based data prefetching. The compiler-based prefetcher [1] needs to reliably predict memory access patterns to insert explicit prefetch instructions or annotate load instructions. This approach is relatively effective in scientific codes which usually have loops containing array accesses whose indices are linear functions of the loop indices. Pure hardware-based prefetch can take advantage of run-time information to dynamically predict prefetch address without the need for programmer or compiler intervention. Jouppi [2] proposed the stream buffer approach. When a cache miss occurs, a stream of K sequential cache blocks from the miss address is brought into a stream buffer. More elaborate hardware prefetching techniques ([3], [4], [5], [6]) have been

proposed following Jouppi's stream buffer work. These approaches consider the non-sequential memory access patterns such as array accesses with large strides or other irregular data access patterns. However, the effectiveness of these prefetchers is limited in general-purpose programs, especially pointer-chasing access codes written in languages such as C and C++.

To address the pointer-chasing and non-stride access patterns, extensive research [7], [8], [9], [10], [11], [12], [13], [14] has studied the correlation between consecutive memory accesses. Chilimbi and Hirzel [8] observed that data reference sequences are frequently repeated in the same order. The earlier work [7] had proposed the use of a Markov predictor to drive a data prefetcher. By dynamically recording sequences of cache miss references in a hardware table, the prefetcher attempts to predict when a previous pattern of misses has begun to repeat itself. When the current cache miss address is found in the table, prefetchers for likely subsequent misses are issued. However, the limitation in the Markov prefetcher is that they attempt to predict correctly only a single miss per prediction, limiting memory level parallelism and prefetch lookahead. More recent prefetchers [13], [10], [14] predict a sequence of successor misses for each current miss. Prior research has also studied the size of the table entry and where the table should be located, either on-chip or off-chip. If the entries of the table is too large, it may lead to inefficient use of the table storage. When the table is too small, it scarifies lookahead and prefetch coverage. In terms of the location, it may not be practical to add the table on-chip since the table may require megabytes of storage. However, moving the table off-chip increases the memory access latencies and the memory bandwidth as well. In [12], it identifies the challenges that makes the prefetchers unattractive and attempts to solve these issues for practical deployment.

III. Level 1 Cache Prefetch

The level 1 cache prefetch (L1P) module in the BG/Q serves as the interface between the core and the rest of the BG/Q system. Each core is paired with a L1P as shown in Figure 1, and any traffic between cores and Level 2 (L2) cache passes through L1P. More details of the BG/Q chip design can be found in [15].

Each L1P maintains 4K byte buffer storing data prefetched from L2. The prefetch line size is 128 byte, a total of 32 lines can be held in the buffer, which are managed by the prefetch directory (PFD). Overall, the structure of the L1P is similar to that of cache. For example, when the buffer is full, the least-recently-used line (LRU) is evicted from the buffer, which is common for caches. The difference may be that the L1P can generate load requests autonomously unlike caches. Two types of prefetch methods, stream prefetch and list prefetch, are provided. The stream prefetch engine is optimized to cope with a linear memory access pattern while the list prefetch is tuned for handling access patterns that are non-uniform (e.g., random) but repeating in some finite range (e.g., The memory access pattern within one loop iteration is random but the memory access patterns across loop iterations are similar).

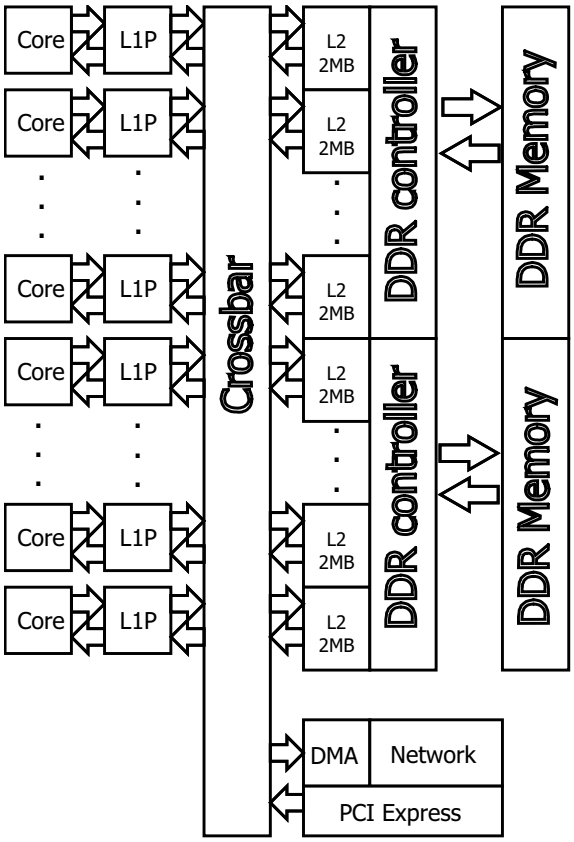


Fig. 1. Blue Gene/Q Memory Hierarchy

These two engines can operate simultaneously, which can serve very well for the applications which require both types of data access pattern. However, the two engines may compete for the shared resources such as the buffer. Interaction between these two engines will be illuminated in the later section.

A. Stream prefetch engine

The stream prefetch engine can prefetch data from a consecutive sequence of 128 byte blocks. The stream engine can maintain up to 16 such sequences, which we will refer each as a stream later on. The engine can automatically establish streams by detecting consecutive memory access pattern or one can force the engine to do so manually.

For automatic establishment, two different policies, confirmed and optimistic, can be used. The concept is illustrated in Figure 2. When the optimistic policy is employed, no consecutive memory access pattern is required to be present and any L1 miss address triggers an establishment of a stream. Such policy should be used only when it is certain that any memory access is consecutive. On the contrary, the confirmed stream establishment policy requires a detection of a stream. In order to detect a stream, the engine uses the stream detection table (SDT). The SDT holds up to 16 addresses. Initially, this table is empty and for any L1 miss, the 128byte aligned address which is considered as the start of a new linear stream,

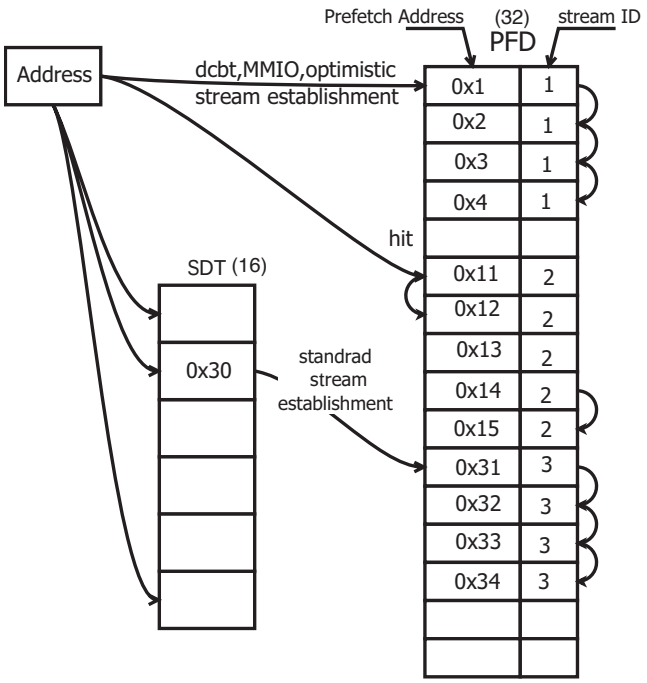


Fig. 2. Blue Gene/Q Stream Prefetch

is registered in the table. When the next L1 miss is present, its address is compared with the entries of the table and if there is a match, it constitutes a detection of a stream. If the address of the new L1 miss does not match any of the entries of the table, this new L1 miss is taken as a new candidate for a start of a new stream and is inserted into the table. If the table is full, a victim is selected via round robin.

Upon an establishment of a stream, user configurable number of addresses are inserted into prefetch directory (PFD) and a stream ID tag is attached to those addresses. The PFD provides the book-keeping of the L1P buffer and sends out an actual data fetch request to the crossbar. It also plays an important role on managing those established streams. When a L1 miss is presented to the L1P, its address is compared with addresses in the PFD. If a match is found, it constitutes a hit for the stream that the matching address belongs to. This hit triggers an advancement of the corresponding stream and so further prefetching for that stream. Also, if the data from the hit address is not ready in the L1P buffer, prefetch-demand fetch conversion occurs and the prefetch depth of the stream is increased so that the data is fetched even earlier. If the PFD is already full, this is done at the expense of decreasing prefetch depth of a stream which is hit least recently.

Under manual stream establishment policy, one can directly establish a stream in the PFD by two different ways. Users can issue "dcbt(data cache block touch)" command and the L1P understands the command and establishes a stream based on the address coming with the "dcbt" command. Or users can write to a special memory mapped I/O (MMIO) register

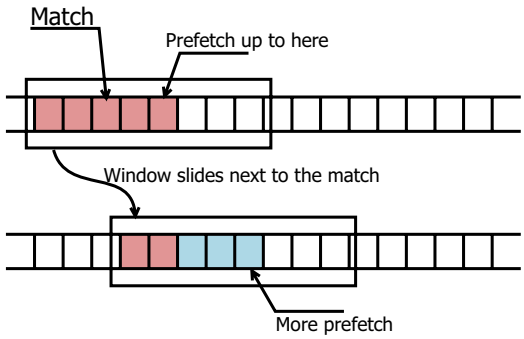


Fig. 3. The Sliding Window for Blue Gene/Q List Prefetch

which directly controls the behavior of the L1P to establish a stream.

The confirmed policy is set by default. Other policy may be selected by a kernel call. Characteristics of those policies are studied in the next section.

B. List prefetch engine

The list prefetch engine prefetches data by a predefined list of addresses. Performing prefetch by a predefined list of addresses requires tracking the location of the current L1 miss in the list. Since it is possible that the memory access pattern may not repeat itself exactly because of the unknown L1 cache state, the list prefetch engine is designed to tolerate such perturbations. The engine maintains an array of size 8, called ReadListArray, which may be understood as a sliding window of the list. Initially, the sliding window focuses on the first 8 addresses of the list. Also, data is prefetched up to a user-configurable depth. When a L1 miss is presented to the L1P, the address is compared with the addresses in the sliding window and if a match is found in a user-configurable range, the sliding window moves next to the match. Simultaneously, this match triggers more prefetch in order to maintain the prefetch depth. If no match is found or the match is out of the range, the sliding window does not move and the mismatch counter is triggered. If there are many consecutive such events, the counter keeps increasing and if it reaches a user-configurable value, the list engine aborts. If there are no such incidents, the sliding window will move toward the end of the list and recognizes the end-of-list mark and the list prefetching finishes. Figure 3 illustrates how the sliding window mechanism works in the list prefetching. The user may monitor the location of the sliding window by looking at the dedicated MMIO register.

Since the predefined list of the addresses resides in the main memory, one can manually create it and pass its starting address to the engine. However, the engine itself provides the functionality of creating the list. The list prefetch engine can store the sequence of the L1 miss addresses in a separate memory space which is allocated by the device driver

while performing prefetching simultaneously. Later, this new sequence of addresses can be used as prefetch addresses. By repeating this, the prefetch accuracy may improve.

A simple pattern compression mechanism is implemented. In particular, it is possible that two L1 misses which fall into the same L1P line since the L1 cache line size is half of the L1P cache line size. The compression mechanism enables creating of a single list entry for those two L1 misses. In order to achieve this, the list compression table (LCT) which contains 4 addresses is used. When there is a match between the L1 miss address and any of the entries of the table, the L1 miss is ignored. The width of an entry of the created list is 32 bits and it contains (part of) the physical address (removing 7 least significant bits for 128-byte alignment), end-of-list marker, and two bits indicating occurrence of upper half and lower half L1P line. As it is stored in the main memory, the created list can be dumped to inspect the memory access pattern for debugging.

The engine can be paused and resumed (e.g., via API function calls). When paused, the list prefetch engine does no prefetching, no movement of sliding window and no creation of list. This function can be used when it is clear that a section of the code does not repeat itself. In such case, the stream prefetch engine may be turned on.

When the list engine is being stopped or being deactivated by a user, it must perform finish-up. For list-creating side, the engine must flush the LCT and attach the end-of-list mark. For list-prefetching side, the engine need to wait until the last memory access is satisfied. This is required for the next activation of the list engine. Hence, the device driver check and makes sure that this is done by polling the status of the list engine.

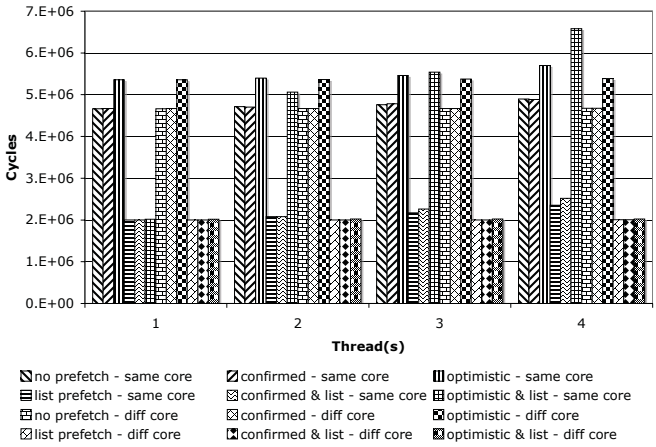
There are four instances of list prefetch engines matching the number of hardware threads and each engine is assigned a different MMIO address space so that each thread can easily and control the matching engine.

IV. Evaluation

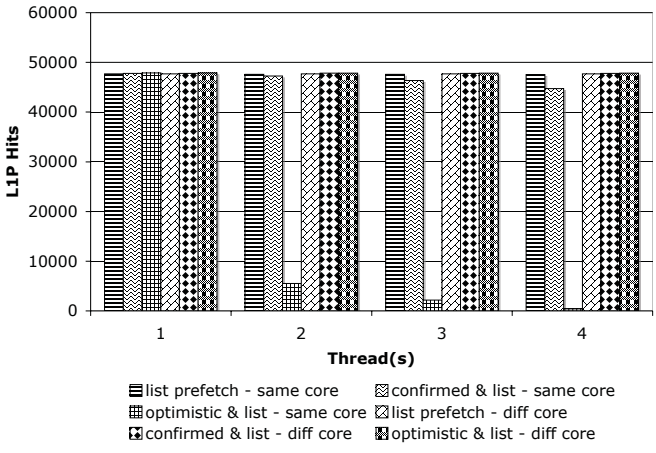
A. Basic Analysis

In this section we use three loops to evaluate the effectiveness of the L1P. The first loop is with only non-uniform memory accesses. The second loop is with half uniform memory accesses and half non-uniform memory accesses. The third loop is with only uniform memory accesses. For each experiment, we use up to four hardware threads on the same core or spreading out onto four cores (i.e., one thread per core up to four cores – so that each thread has all the resource on the core) with different prefetch mechanisms. "no prefetch" is with L1P turned off. "confirmed" and "optimistic" are the policies for the stream prefetcher and "list" is to use the list prefetcher. The "Time performance" measures the execution time (or cycles) using the time based register in the hardware counter and lower is better along the y-axis. The "L1P Hits" is the number of hits in the L1P buffer which higher is better.

Figure 4 shows the result for the first loop with only non-uniform memory accesses. The loop is within a benchmark with weak scaling workload for threads. In Figure 4(a), it shows the stream prefetcher is not helpful with non-uniform memory accesses. List prefetcher improves the performance when there is enough resource (e.g., L1P buffer) for the hardware threads. However, if the stream prefetcher is too aggressive (e.g., 6th bar of each set: optimistic & list - same core; one set of bars per each thread number) and is competing with the list prefetcher for the resource (e.g., L1P buffer), it will cause a negative impact to the performance. This is verified by the number of L1P hits in Figure 4(b) (3rd bar of each set).



(a) Time performance

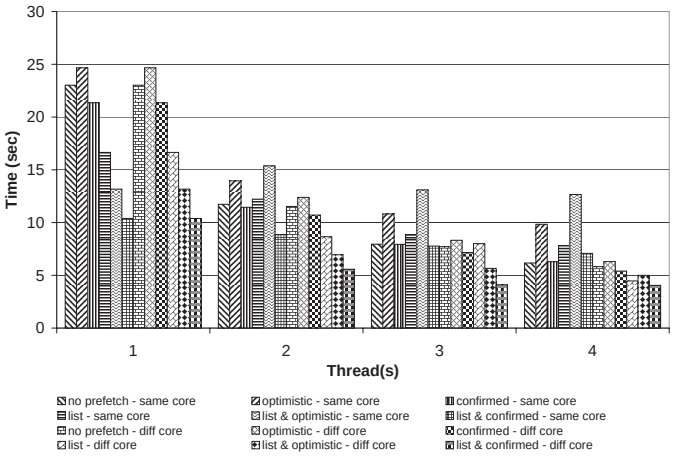


(b) L1P Hits

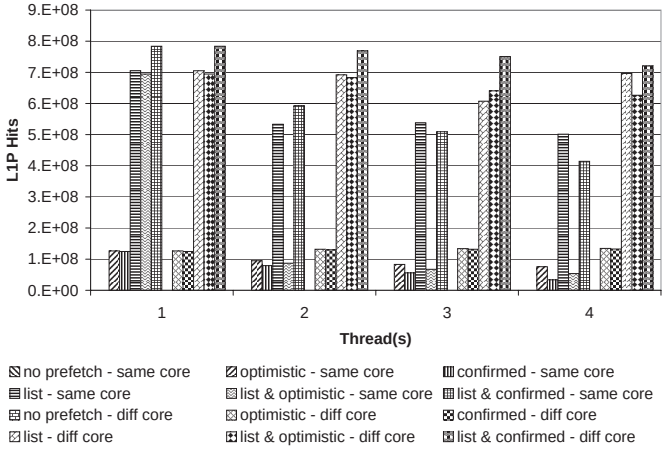
Fig. 4. Non-uniform memory accesses with different prefetch mechanisms

Figure 5 shows the result for the second loop with half uniform and half non-uniform memory accesses. The loop is part of an application with strong scaling workload for threads. As shown in Figure 5(a), the list prefetcher does a reasonably good job to prefetch the needed data into L1P. The performance can be further improved if working together with the stream prefetcher using confirmed policy. When stream prefetcher is also on with optimistic policy, and the resource is limited (e.g., two or more threads on the same core), the L1P hits number drops and cause negative impact to the

performance (the 5th bar of each set: list & optimistic - same core in Figure 5(a) and Figure 5(b)[1]).

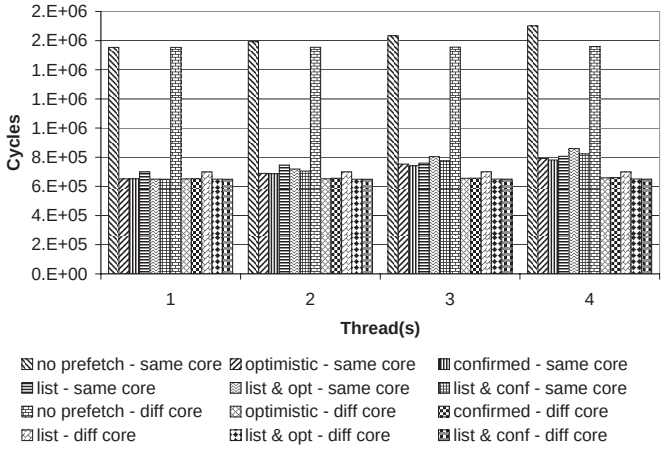

Time (sec)
Thread(s)
no prefetch - same core
optimistic - same core
confirmed - same core
list - same core
list & optimistic - same core
list & confirmed - same core
no prefetch - diff core
optimistic - diff core
confirmed - diff core
list - diff core
list & optimistic - diff core
list & confirmed - diff core

(a) Time performance

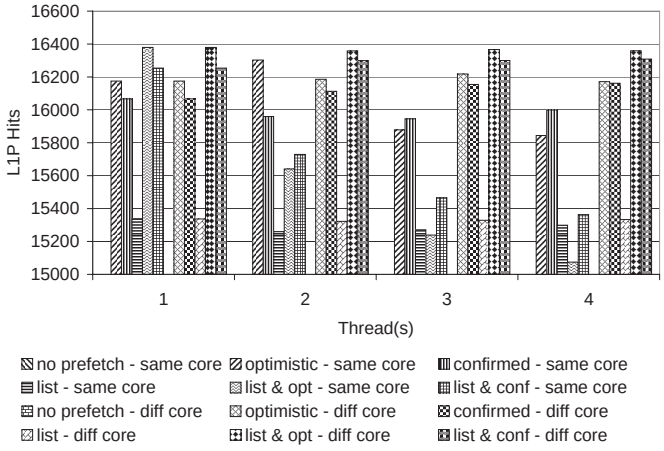

L1P Hits
Thread(s)
no prefetch - same core
optimistic - same core
confirmed - same core
list - same core
list & optimistic - same core
list & confirmed - same core
no prefetch - diff core
optimistic - diff core
confirmed - diff core
list - diff core
list & optimistic - diff core
list & confirmed - diff core

(b) L1P Hits

Fig. 5. Mixed memory accesses with different prefetch mechanisms


Cycles
Thread(s)
no prefetch - same core
optimistic - same core
confirmed - same core
list - same core
list & opt - same core
list & conf - same core
no prefetch - diff core
optimistic - diff core
confirmed - diff core
list - diff core
list & opt - diff core
list & conf - diff core

(a) Time performance


L1P Hits
Thread(s)
no prefetch - same core
optimistic - same core
confirmed - same core
list - same core
list & opt - same core
list & conf - same core
no prefetch - diff core
optimistic - diff core
confirmed - diff core
list - diff core
list & opt - diff core
list & conf - diff core

(b) L1P Hits

Fig. 6. Stride-one memory accesses with different prefetch mechanisms

The third loop result is shown in Figure 6. It is with only uniform memory accesses and it is with weak scaling workload for threads. The result in Figure 6(a) shows that the list prefetcher is competitive to the stream prefetcher to achieve similar performance (2nd, 3rd and 4th bar of each set in Figure 6(a)). The performance is also reasonably good when both prefetchers are turned on (5th and 6th bar of each set in Figure 6(a)).

The last three bars from each set in the figures (diff core) are the cases when only one hardware thread is used on each core. This shows when there is sufficient amount of resource, the stream and list prefetch engines may cooperate well to achieve good performance.

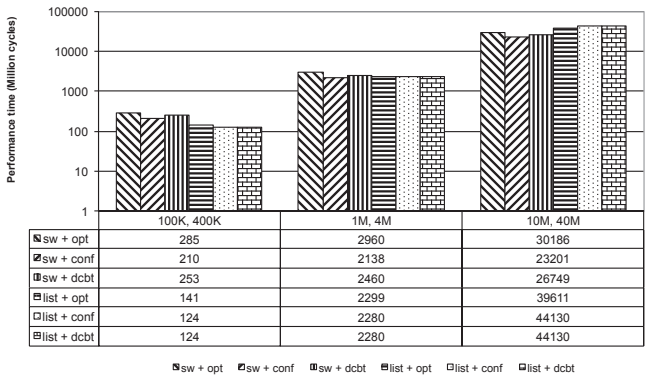
B. Graph Algorithm

We study the impact of various prefetching strategies on BG/Q for a graph algorithm implementation. We implement a variant of the Shiloach-Vishkin algorithm (SV) for the connected components problem with large, sparse inputs. SV is representative of several connectivity algorithms in that it adapts the widely-used graft-and-shortcut approach. Through carefully designed grafting schemes, the algorithm achieves complexities of $O[\log n]$ time and $O[(m+n)\log n]$ work under the arbitrary CRCW PRAM model. It can be extended naturally to solve the spanning tree problem under the priority CRCW PRAM model with the same complexity bound.

For graph $G = (V, E)$ ($|V| = n$, $|E| = m$) represented as an edge list, SV starts with n isolated vertices and m processors. Array D of size n stores the connected components that a vertex belongs. $D[i] = i$ $(0 \le i \le n-1)$ initially. For edge $e_i = (u, v)$, processor P_i $(0 \le i \le m-1)$ inspects $D[u]$ and $D[v]$, and if $D[v] < D[u]$, it grafts component $D[u]$ to $D[v]$; if $D[u] < D[v]$, it grafts component $D[v]$ to $D[u]$. Grafting created rooted trees, and these trees are rooted stars using pointer-jumping at the shortcut step. The algorithm iterates until no further grafting is possible.

The grafting of components dominates the execution time. One iteration of grafting is shown below.

[1]The 1st and 7th bars of each set in L1P Hits (Figure 5(b) and Figure 6(b)) are 0's since there is no prefetch.



	100K, 400K	1M, 4M	10M, 40M
sw + opt	285	2960	30186
sw + conf	210	2138	23201
sw + dcbt	253	2460	26749
list + opt	141	2299	39611
list + conf	124	2280	44130
list + dcbt	124	2280	44130

Fig. 7. Performance impact of various prefetchings

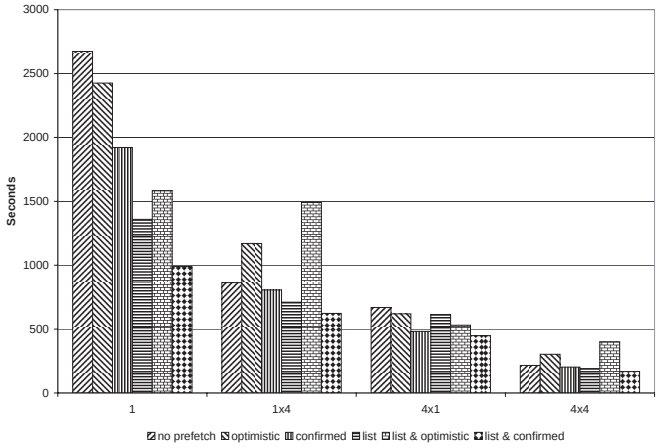


Fig. 8. NPB CG with various prefetchings

```
parallel_do(i=0; i<m; i++)
{
        u = El[i].v1; v = El[i].v2;
        if(D[u]<D[v])
        {
                D[D[v]]=D[u];
        }
        if(D[v]<D[u])
        {
                D[D[u]]=D[v];
        }
}
```

If we inspect the memory access pattern in the grafting loop, we see that access to the edge array *El* is sequential, while access to the component array *D* is random if the input is a large, irregular graph. Sequential accesses benefit from stream prefetching, while random access may benefit from list prefetching as between consecutive grafting iterations there are similarities in the access pattern.

Besides hardware prefetching engines, software prefetching may also be applied to prefetch the irregular access. The grafting loop may be tiled with tile size k, $1 \leq k \leq m$. The inspection of the edge list proceeds in blocks of size k. So instead of directly accessing the D array in parallel, we first prefetch $2k$ of the D values in a prefetch loop. Then we conduct grafting for the k edges in the hope that the required value has already been brought into cache by the previous prefetching loop. We also study the impact of software prefetching on the performance.

Figure 7 shows the performance impact of combining software prefetching with different stream prefetching schemes and combining list prefetching with different stream prefetching schemes. We test on three random graphs with different sizes, that is, 100K vertices and 400K edges, 1M vertices and 4M edges, and 10M vertices and 40M edges.

For the input size 100K vertices, the list prefetching shows much better performance. The software prefetching may work similarly well but the cost of issuing prefetching commands such as "dcbt" slows down the code. However, the list prefetching loses its advantage when the input size becomes large. At 1M vertices, the size of list itself of addresses is bigger than 32MB, the size of L2. Since the list cannot be contained in L2, it has to be brought from the main memory. Also, the list and the component array *D* fight for L2 resources. So, the performance improvement by the list prefetching is lost. For even larger input size, 10M vertices, this effects becomes more significant and the performance gets even worse.

Since the list perfetching improves the performance only for small graphs, one may break the big graph into small pieces and handle one by one and then combine them later. This requires clever algorithms of dividing the big graph properly, which will be our future research direction.

C. Conjugate Gradient iteration and sparse matrix

The conjugate gradient (CG) method is the one of the most frequently used algorithms to solve linear systems. For example, a discretized partial differential equation is involved with a sparse matrix and its solution can be obtained by the CG method. The CG method requires iteration of sparse matrix and vector multiplication. Since the sparse matrix is represented by CSR format (compressed sparse row), when a matrix vector multiplication is performed, access pattern of the elements of the vector is essentially random. Such memory access may be efficiently prefetched by the list prefetch while conventional stream prefetch may fail. In this section, we show the performance result using class C of the CG OpenMP version 3.3 in the NAS Parallel Benchmarks [16] with various prefetching schemes.

The kernel loop we instrumented in the CG consists 60% of the strided memory access pattern and 40% of a random memory access pattern. Accessing the matrix elements(8Byte) and the column indices (4Byte) in the CSR format makes the uniform pattern while accessing the vector elements(8Byte) by the column indices generates the random pattern. Figure 8 shows the performance results. We experiment with three different threading and core arrangements. For the legend, the

"1" means one single thread. The $a \times b$ means a core(s) with b thread(s) on each core (e.g., "4x1" means four cores with one thread on each core). The "no prefetch" means no prefetching and the rest are various prefetching mechanisms. The loop is with strong scaling so for the same workload, it is expected with n threads, the execution time should $1/n$ of that with 1 thread.

As shown in the figure, the list prefetch together with the confirmed stream prefetch achieve reasonably good performance results. The optimistic stream prefetch is being aggressive and does not achieve good performance compared to the confirmed stream prefetch. This is because with the optimistic stream prefetch, the stream prefetch engine generates too many data requests (that may not be useful) and delay the data requests (that are actually needed). With limited resource such as 1×4 and 4×4, the performance result of the optimistic stream prefetch is worse than that with no prefetch. This is again due to the aggressiveness of the optimistic prefetch policy.

V. Discussion

The L1P prefetches the data from L2 into L1P based on the fixed (strided) and/or recorded (random) memory access patterns. This helps to improve the performance if the there is sufficient resource and time to prefetch the data (e.g., enough work to do in the loop to overlap the computation with the data prefetching). As shown in the Table I, if the data requests from the L1 misses are predicted correctly by the L1P prefetch engines, it can reduce 2/3 of the latency. However, there are still about 18 cycles latency from L1P into L1 that has to be overlapped (e.g., with computation) to further improve the performance.

L1	6 cycles
L1P	24 cycles
L2	84 cycles
Memory	346 cycles

TABLE I
MEMORY HIERARCHY LATENCY

For the L1P list prefetch engine to record the addresses, each 128-byte block address is a 4-byte data in the prefetch list. The stream of prefetch list addresses will go through L2 and be stored in the main memory. This will have impacts on the L2 and introduce traffic to the main memory. The size of the list in the main memory is normally set to be the number of L1 misses (from the hint of the hardware counter) or set by the user. The list prefetch reduces the L1 cache miss penalty at the cost of the memory bandwidth (to L2 and/or main memory) and the storage space (in L2 and main memory for addresses). Unless the application is heavily memory-bound, the use of the list prefetching should incur little overhead to the application execution.

The 4K bytes of the L1P buffer is shared between the stream and list prefetch engines up to four hardware threads. As shown in the experiment results, when more prefetch engines are sharing the L1P buffer, the competition for the buffer entry space may cause thrashing. Without enough resource, the situation may get worse when the core is packed with multiple threads.

Currently the size of the window for the list prefetch engine deciding whether the data address requested from the L1 miss is hit or not is 8. This window size decides the number of the mismatch between two streams of addresses (representing two memory access patterns) allowed. As we learned from some applications like LAMMPS [17], the memory access patterns are similar from one to another. However the length of consecutive mismatch can exceed 8 easily. This will cause the list prefetch to be abandoned. One solution is to increase the window size. When the window size is constrained by the hardware (e.g., parallel comparison in the given number of cycles), one possible solution is to break the long list into multiple small lists. This will require more system software support, user hints or automatic detection from the performance tools.

The lists of addresses stored by the list prefetch engine provide a possible channel for performance tuning people to understand the application execution behavior. Together with the hardware counter events and the debugging facility in the core, we can study the memory access pattern for the application such as how and when the memory access pattern changes. For our future work, we plan to further investigate using this list information.

VI. Conclusion

In this paper, we have presented a description and evaluation of the new Level One Cache Prefetch (L1P) feature of the Blue Gene/Q supercomputer.

This L1P feature is intended to benefit applications that have traditionally been limited in their performance due to indirect or random memory access patterns. The Blue Gene/Q L1P feature has been demonstrated here to be effective on graph-based applications and sparse matrix solvers provided that sufficient L1P buffer resources are available, i.e., for small contention of this buffer. Also, there can be synergistic performance improvement in conjunction with stream prefetching when data sizes are small enough to fit within the L2 cache. More intelligent logic for enabling and disabling this feature in real time is a current area of research, as is with the coordination of massive speculative threading.

VII. Acknowledgments

We would like to thank Robert Walkup, Ravi Nair, Kristan Davis, Bruce D'Amora, John Magerlein, David Klepacki and Alan Gara for their generous support and advice.

References

[1] T. Mowry, M. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operation Systems*, September 1992.

[2] N. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of the Seventeenth International Symposium on Computer Architecture*, May 1990.
[3] S. Palacharla and R. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.
[4] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic, "Memory-system design considerations for dynamically-scheduled processors," in *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.
[5] T. Sherwood, S. Sair, and B. Calder, "Predictor-directed stream buffers," in *Proceddings of the 33rd IEEE/ACM International Symposium on Microarchitecture*, 2000.
[6] C. Zhang and S. McKee, "Hardware-only stream prefetching and dynamica access ordering," in *Proceedings of the 14th International Conference on Supercomputing*, 2000.
[7] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *Proceedings of the 24th International Symposium on Computer Architecture*, 1997.
[8] T. Chilimbi, "Efficient representation and abstractions for qualifying and exploiting data reference locality," in *Proceedings of the ACM SIGPLAN Conference on Programming Lanuguage Design and Implementation*, 2001.
[9] T. Chilimbi and M. Hirzel, "Dynamic hot data stream prefetching for general-purpose programs," in *Proceedings of the ACM SIGPLAN Conference on Programming Lanuguage Design and Implementation*, 2002.
[10] T. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi, "Temporal streaming of shared memory," in *Proceedings of the 32nd International Symposium on Computer Architecture*, 2005.
[11] T. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal streams in commercial server applications," in *IEEE International Symposium on Workload Characterization*, 2008.
[12] ——, "Practical off-chip meta-data for temporal memory streaming," in *The 15th International Symposium on High-Performance Computer Architecture*, 2009.
[13] K. Nesbit and J. Smith, "Data cache prefetching using a global history buffer," in *Proceedings of the 10th Symposium on High-Performance Computer Architecture*, 2004.
[14] Y. Chou, "Low-cost epoch-based correlation prefetching for commercial applications," in *Proceddings of the 40th IEEE/ACM International Symposium on Microarchitecture*, 2007.
[15] R. A. Haring, M. Ohmacht, T. W. Fox, M. K. Gschwind, D. L. Satterfield, K. Sugavanam, P. W. Coteus, P. Heidelberger, M. A. Blumrich, R. W. Wisniewski, A. Gara, G. L.-T. Chiu, P. A. Boyle, N. H. Chist, and C. Kim, "The ibm blue gene/q compute chip," vol. 32. Los Alamitos, CA, USA: IEEE Computer Society, 2012, pp. 48–60.
[16] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, and L. Dagum, "The nas parallel benchmarks," *International Journal of High Performance Computing Applications*, 1994.
[17] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of Computational Physics*, vol. 117, pp. 1–19, 1995.